A Flexible Design Automation Tool for Accelerating Quantized Spectral CNNs

Rachit Rajat*
University of Southern California
rrajat@usc.edu

Hanqing Zeng*
University of Southern California
zengh@usc.edu

Viktor Prasanna
University of Southern California
prasanna@usc.edu

***Abstract*—CNNs have proven to be extremely powerful in various computer vision applications. To alleviate the computation burden and improve hardware efficiency, low-complexity convolution algorithms (e.g., spectral convolution) and data quantization schemes have been implemented on FPGAs. However, to translate the reduced algorithm complexity into improved hardware performance, we need significant manual tuning of mapping parameters specific to the CNN model and the target FPGA device. We propose a flexible tool to automate the process of generating high throughput accelerators for *quantized, spectral* CNNs. The tool takes as input high level specification of the CNN model, the data quantization scheme and the target hardware architecture. It outputs synthesizable `Verilog` after fast exploration of the complete design space. Our tool is flexible in three dimensions: 1) data representation, 2) FPGA architecture, and 3) CNN models. To support arbitrary quantization bit width, we propose a resource-efficient multiplier design, which uses the fixed, high bit-width DSPs to implement various low bit-width complex multiplications needed in spectral CNNs. To support FPGAs with limited on-chip memory, we propose a systolic array-based architecture for spectral convolution, which exploits high computation parallelism in DSPs without stressing BRAM resources. To support CNNs with various layer parameters, we tile and permute data blocks to saturate the communication and computation capacity. Finally, we propose a fast design space exploration algorithm to complete the end-to-end `Verilog` generation. The whole design space exploration and verilog generation takes less than 1 second on an Intel Core i5 laptop. We perform evaluation on Stratix-10 and Stratix-V FPGAs, using AlexNet and VGG16. The generated accelerators achieve $2\times$ to $4\times$ higher throughput than state-of-the-art, for 8-bit and 16-bit data quantization.**

I. Introduction

Convolutional Neural Networks (CNNs) are powerful deep learning models widely used in the field of computer vision [1], [2]. The following challenges exist while performing inference on FPGA:

1) Computation load: sliding window type of spatial convolution requires large number of operations.
2) Design effort: manual hardware implementation and design space exploration is time consuming.

For the first challenge, accelerators designed for spectral CNNs (using frequency domain convolution) have been implemented on FPGAs [3]–[5]. The designs demonstrate significantly higher throughput (up to $5\times$) than the spatial CNN accelerators, due to the decrease in arithmetic operations. The work in [6] further proposes data quantization to address the drawback of increased model size (due to enlarged convolutional kernels) in the spectral CNN. Overall, quantized spectral CNNs demonstrate the ability to overcome the challenge of high computational requirement of spatial CNNs.

As for the second challenge, various design automation tools [7]–[10] have been proposed to generate high performance inference accelerators. While these efforts significantly alleviate the burden on hardware designers, they face various limitations as well. Most design automation tools [7], [9] target computation kernels of spatial convolution. And the only tool generating spectral CNN accelerators [8] does not support low bit-width arithmetic operations. In addition, the pipeline generated in [8] results in high pressure on BRAMs to support the DSP computation. Such designs may suffer from low clock speed on FPGAs with massive DSP resources [11].

We propose a tool to generate high throughput CNN inference accelerators which solves the aforementioned problems. Our tool is inspired by the performance analysis for spectral quantized CNNs [3], [6].

This paper makes the following contributions:

- We propose an automation tool for quantized spectral CNN accelerators. The tool has the following flexibility:
  - *Quantization schemes*: We propose a low overhead implementation for low bit-width complex multiplication. Our design utilizes the built-in high bit-width DSPs to support a wide range of quantization schemes.
  - *FPGA architecture*: To achieve high throughput under BRAM resource constraints, we reduce spectral convolution to matrix multiplication, and parallelize by systolic arrays. The corresponding accelerator can scale to large FPGA devices without clock rate degradation.
  - *CNN models*: To avoid expensive runtime reconfiguration on convolution layers of various model parameters, we perform data tiling and data permutation to ensure high utilization of the computation pipeline.
- The tool generates the complete inference accelerator in synthesizable `Verilog`. The fast design space exploration algorithm derived from a precise performance model quickly identifies the optimal design point.
- We evaluate our tool on various FPGAs (Stratix-10, Stratix-V), CNNs (AlexNet, VGG16) and data quantization schemes (16-bit to 2-bit). Compared with state-of-the-art spectral and spatial CNN implementation, our designs consistently operate under high clock rate and achieve $2\times$ to $4\times$ higher throughput.

*Equal contribution

This work was supported in part by Intel Strategic Research Alliance and in part by National Science Foundation award number CNS-1643351.

II. Background and Related Work

A. Spectral CNNs

Convolutional layers are the focus of this work. For a spatial convolutional layer handling a batch of b images, its i^{th} input (output) activation, $\boldsymbol{X}_i$ ($\boldsymbol{Y}_i$), has c_{in} (c_{out}) channels, where each channel is a $h_{\text{act}} \times h_{\text{act}}$ 2D matrix. The layer weights $\boldsymbol{W}$ contain $c_{\text{out}} \times c_{\text{in}}$ number of spatial kernels, each being a small $h_{\text{krn}} \times h_{\text{krn}}$ 2D matrix. We can use Fast Fourier Transform (FFT) to convert a spatial layer to its equivalent form in spectral domain. Use $\mathcal{F}(\cdot)$ and $\mathcal{F}^{-1}(\cdot)$ to denote the 2D-FFT operation and its inverse. Thus, the spectral weights $\widetilde{\boldsymbol{W}}_{k,j} = \mathcal{F}(\boldsymbol{W}_{k,j})$, where $1 \le k \le c_{\text{out}}$ and $1 \le j \le c_{\text{in}}$. The spectral input activation $\widetilde{\boldsymbol{X}}_{i,j} = \mathcal{F}(\boldsymbol{X}_{i,j})$, where $1 \le i \le b$ and $1 \le j \le c_{\text{in}}$. A convolutional layer operates as follow:

$$\boldsymbol{Y}_{i,k} = \sum_{j=1}^{c_{\text{in}}} \boldsymbol{X}_{i,j} * \boldsymbol{W}_{k,j} = \mathcal{F}^{-1}\left(\sum_{j=1}^{c_{\text{in}}} \widetilde{\boldsymbol{X}}_{i,j} \circ \widetilde{\boldsymbol{W}}_{k,j}\right) \tag{1}$$

where "$*$" denotes spatial convolution (sliding window); "$\circ$" denotes 2D Hadamard product (element-wise multiplication).

Equation 1 holds whenever the FFT size n is larger than the spatial kernel size h_{krn}. Partitioning or padding of $\boldsymbol{X}$ may be required (Overlap-Add technique [3]) when sizes of n and h_{act} do not match. The value of n affects the overall computation complexity of the spectral CNN, as well as the hardware communication and computation load-balance. Thus, n should be carefully chosen given the target CNN and FPGA.

Spectral CNNs have been implemented on FPGAs as a faster alternative of spatial CNNs. [5] compares both variants with comprehensive theoretical analyses. [3] shows that using spectral convolution algorithm reduces the overall number of arithmetic operations by $5\times$ for large CNNs. [12] proposes a hybrid FPGA implementation combining Winograd-based and FFT-based convolution algorithms for layers of large kernels.

B. Data Quantization

Fixed-point data quantization on CNNs has been widely explored. Bit width to represent the values of $\boldsymbol{W}$ and $\boldsymbol{X}$ can be dramatically reduced with suitable quantization and re-training algorithms. Quantization by the pioneering work [13] achieves high accuracy on a digit recognition dataset with 1- to 4-bit fixed-point numbers. [14] allocates the number of bits according to the characteristic of each layer. Its algorithm works well for AlexNet [1] and VGG16 [2]. [15], [16] explore the potential of fully binarized CNNs. Binary representation of weights, while increasing inference speed-up, may lead to significant accuracy drop for large dataset and CNN models.

As for spectral CNNs, [12] quantizes spectral weights $\widetilde{\boldsymbol{W}}$ to 8 bits, without reporting accuracy. [6] proposes a theoretical framework to systematically quantize $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{X}}$.

C. Design Automation Tools

To facilitate inference accelerator development, design automation tools have been proposed. In [7], synthesis techniques have been proposed for systolic array-based architecture. However, to the best of our knowledge, systolic array-based designs have not been used for spectral CNNs yet. In [10], a RTL compiler has been proposed to accelerate all layers of a CNN. [17] proposes a framework for mapping CNNs onto FPGAs by RTL-HLS templates. Recent work [9] proposes a highly optimized automation tool for generating high throughput inference engines, by intelligent resource allocation algorithms. Note that, all the above works target spatial CNNs only.

[3], [8] present design automation tools for spectral CNNs. The tool by [8] uses a fast code generation algorithm by decomposition of the large design space. However, the algorithm selects the FFT size n to only minimize operation count of spectral convolution. The actual hardware performance may not be optimal, due to communication-computation load-imbalance on FPGAs. The tool by [3] further improves upon [8]. It proposes first-order approximations to the performance model, so that fast design space exploration directly optimizes inference throughput. Although the tool generates high quality designs on small devices such as Stratix V, its architecture may not scale to larger FPGAs, due to the parallelization strategy on the tensors $\widetilde{\boldsymbol{X}}$ and $\widetilde{\boldsymbol{W}}$ (see details in Section III-B). Moreover, both [3], [8] only support 16-bit quantization on $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{X}}$.

In summary, there is an urgent need for a flexible design automation tool targeting quantized spectral CNNs. The tools for spatial CNNs cannot generate spectral CNN accelerators due to the different computation and data flow patterns. The existing tools for spectral CNNs are not flexible to handle various quantization schemes and hardware constraints.

III. Tool for Quantized Spectral CNNs

A. Overview

Our tool takes as input the specification of a pre-trained spectral CNN with quantized weights, the FPGA architecture and the quantization scheme. The tool generates synthesizable `Verilog` for the inference hardware based on an architecture template (Section III-B). Leveraging fast design space exploration (Section III-C), the tool identifies the optimal mapping parameters (including data parallelism of hardware modules, batch size and data tiling factor). A `Python` script assembles the hardware modules into a complete inference pipeline.

In our model, target FPGA architecture is specified by:

- DSP parameters Q_{P}, $\mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}$: Q_{P} defines bit width supported by the multiplier. $\mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}$ defines the total number of Q_{P}-bit multipliers on chip. For example, in Stratix 10 GX2800 [11], $\mathcal{N}_{\text{DSP}}^{18} = 11520$ or $\mathcal{N}_{\text{DSP}}^{27} = 5760$ for 18-bit or 27-bit multiplication supported by the DSP blocks.
- BRAM parameters Q_{B}, $\mathcal{N}_{\text{BRAM}}^{Q_{\text{B}}}$: Q_{B} defines the width of a BRAM block (bits per row). $\mathcal{N}_{\text{BRAM}}^{Q_{\text{B}}}$ defines the number of width-Q_{B} BRAM blocks on chip. For Stratix-10 GX2800 with M20K BRAMs, $\mathcal{N}_{\text{BRAM}}^{20} = 11721$.
- External DRAM parameters Q_{M}, $\mathcal{N}_{\text{DRAM}}^{Q_{\text{M}}}$: For example, $\mathcal{N}_{\text{DRAM}}^{16} = 8$ means in each FPGA clock cycle, eight 16-bit words can be read from or written to DRAM.

Each convolutional layer of the CNN is specified by:

- h_{act} — spatial dimensions of each layer's activation $\boldsymbol{X}$.
- h_{krn} — spatial dimensions of each layer's kernel map $\boldsymbol{W}$.

- c_{in} — number of input channels of each layer.
- c_{out} — number of output channels of each layer.
- n — 2D FFT size to transform $\boldsymbol{X}$ to $\widetilde{\boldsymbol{X}}$.

Finally, the quantization scheme is specified by:

- q_{act} — quantization bit width for spatial activation $\boldsymbol{X}$.
- $\widetilde{q}_{\text{act}}$ — quantization bit width for spectral activation $\widetilde{\boldsymbol{X}}$.
- $\widetilde{q}_{\text{krn}}$ — quantization bit width for spectral weights $\widetilde{\boldsymbol{W}}$.

B. Spectral Convolution Engine

Parallelization strategies targeting different dimensions of the tensors $\widetilde{\boldsymbol{X}}$, $\widetilde{\boldsymbol{W}}$ result in various computation and communication tradeoffs. Our analysis leads to a flexible architecture template that sustains high throughput on a variety of CNNs, FPGAs and quantization schemes. Starting from Equation 1, the operations by a spectral convolutional layer are[1]:

$$\texttt{OP}_1: \ \widetilde{\boldsymbol{X}}_{i,j} = \mathcal{F}\left(\boldsymbol{X}_{i,j}\right), \qquad \forall i \in [1,b], j \in [1, c_{\text{in}}]$$

$$\texttt{OP}_2: \ \widetilde{\boldsymbol{Y}}_{i,k} = \sum_{j=1}^{c_{\text{in}}} \widetilde{\boldsymbol{X}}_{i,j} \circ \widetilde{\boldsymbol{W}}_{k,j}, \qquad \forall i \in [1,b], k \in [1, c_{\text{out}}]$$

$$\texttt{OP}_3: \ \boldsymbol{Y}_{i,k} = \mathcal{F}^{-1}\left(\widetilde{\boldsymbol{Y}}_{i,k}\right), \qquad \forall i \in [1,b], k \in [1, c_{\text{out}}]$$

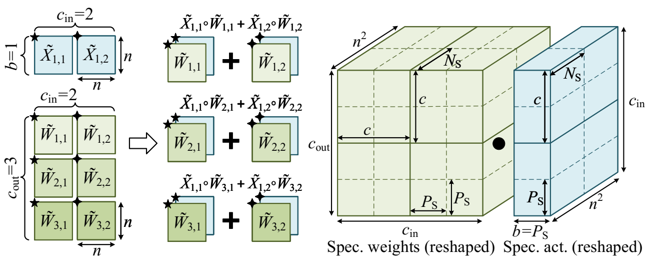
From [3], among the three operations, $\texttt{OP}_2$ dominates the computation workload. Below we present two different interpretations (correspondingly, two parallelization strategies) of $\texttt{OP}_2$. Figure 1a shows a more intuitive view using Hadamard product as the computation primitive. One layer requires $c_{\text{in}} \cdot c_{\text{out}}$ number of $n \times n$ Hadamard products, each operating on a $\left(\widetilde{\boldsymbol{X}}_{i,j}, \widetilde{\boldsymbol{W}}_{k,j}\right)$ pair. Since Hadamard product incurs no dependency among the pixels of $\widetilde{\boldsymbol{X}}$ and $\widetilde{\boldsymbol{W}}$, a 1D multiplier array may effectively parallelize $\texttt{OP}_2$ by unfolding the $n \times n$ dimensions [3], [8]. In Figure 1a, $\left(1 \cdot 2 \cdot 3 \cdot n^2\right)$ number of multipliers can finish $\texttt{OP}_2$ in one cycle. Unfortunately, such straightforward parallelization strategy does not work well for state-of-the-art large FPGAs. On one hand, the lack of data dependency in the Hadamard product simplifies the control logic and data flow. On the other hand, it results in high pressure on BRAMs to meet the data parallelism requirement. In Figure 1a, each cycle, the multiplier array initiates $(2 \cdot 1 + 2 \cdot 3) \cdot n^2$ distinct reads into BRAMs. On an FPGA with thousands of DSPs [11], the data parallelism to be supported by BRAMs is in the order of thousand to ten thousand. Such a design may suffer from clock rate degradation and low BRAM utilization.

To reduce the pressure on BRAMs, we parallelize $\texttt{OP}_2$ along other dimensions (i.e., channels c_{in}, c_{out} and batch b). In Figure 1b, $\texttt{OP}_2$ on the spectral weights and activations is computed using dot product (instead of Hadamard product) as the primitive. The following are equivalent to compute $\widetilde{\boldsymbol{Y}} \in \mathbb{R}^{b \times c_{\text{out}} \times n \times n}$:

$$\widetilde{\boldsymbol{Y}}_{i,k,\cdot,\cdot} = \sum_{j=1}^{c_{\text{in}}} \left(\widetilde{\boldsymbol{X}}_{i,j} \circ \widetilde{\boldsymbol{W}}_{k,j}\right), \quad \forall i \in [1,b], k \in [1, c_{\text{out}}] \quad \text{(2a)}$$

$$\widetilde{\boldsymbol{Y}}_{\cdot,\cdot,d,e} = \left(\widetilde{\boldsymbol{W}}_{\cdot,\cdot,d,e}\right) \cdot \left(\widetilde{\boldsymbol{X}}_{\cdot,\cdot,d,e}\right)^T, \qquad \forall d,e \in [1,n] \quad \text{(2b)}$$

[1]Note that since the kernel weights $\boldsymbol{W}$ and $\widetilde{\boldsymbol{W}}$ are fixed during inference, we prepare $\widetilde{\boldsymbol{W}}$ offline and do not consider the cost of $\widetilde{\boldsymbol{W}} = \mathcal{F}\left(\boldsymbol{W}\right)$.



(a) Expressed as Hadamard (b) Expressed as dot product

Fig. 1: Two views of spectral convolution of a layer

In Equation 2b, $\texttt{OP}_2$ is decomposed into n^2 number of (independent) dot products between $c_{\text{out}} \times c_{\text{in}}$ matrices and $c_{\text{in}} \times b$ matrices. Matrix dot product can be efficiently implemented on FPGAs by systolic arrays. Note that a 2D $N \times N$ systolic array performs $N \times N$ MAC operations each cycle, while requiring only $2N$ data from on-chip memory. Thus, comparing Design 1 (from Equation 2a) and Design 2 (from Equation 2b):

- Both have simple dataflow and low logic overhead.
- Design 2 reduces the needed data parallelism from BRAMs by orders of magnitude, compared with Design 1.

We describe the architecture based on Equation 2b below.

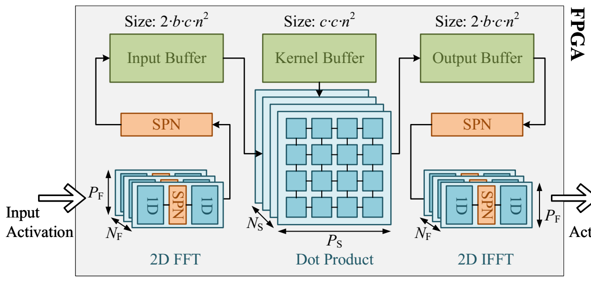


Fig. 2: Overview of the spectral convolution engine

As shown in Figure 2, the pipeline in the spectral convolution engine consists of three memory modules (*input buffer*, *kernel buffer* and *output buffer*), three computation modules (*2D FFT*, *dot product* and *2D IFFT*) and four data *permutation network*. As in overview, a tile of spatial activations $\boldsymbol{X}$ is streamed into the 2D FFT module from external DRAM. Output of 2D FFT (i.e., $\widetilde{\boldsymbol{X}}$) is permutated and then stored into the corresponding memory banks of the input buffer. Majority of computation is performed by the dot product module. The module consists of multiple equal-size 2D systolic arrays. Each cycle, these systolic arrays read data from input and kernel buffers, compute $\widetilde{\boldsymbol{Y}}$ and store outputs in the output buffer. Finally, the 2D IFFT module reads $\widetilde{\boldsymbol{Y}}$ from output buffer in a streaming fashion, computes the inverse Fourier transform, and transfers the result $\boldsymbol{Y}$ back to external DRAM. Note that we use double buffering on input and output buffers to overlap the external DRAM communication with the on-chip computation.

2D FFT / IFFT module: $n \times n$ 2D FFT can be decomposed into n n-point 1D FFTs (row major), followed by another n number of n-point 1D FFTs (column major). We use the 1D FFT pipeline from [18]. Between the two stages of 1D

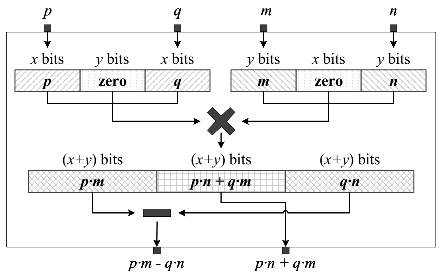


Fig. 3: Multiplier for quantized complex numbers

FFT, we use a Streaming Permutation Network (SPN) [19] to transpose the $n \times n$ intermediate result. We parameterize the data parallelism of the 1D FFT pipelines as P_{F}. We also allow the module to process N_{F} number of $n \times n$ 2D FFTs in parallel, corresponding to data from different input channels.

Dot product module: This module deploys N_{S} number of $P_{\text{S}} \times P_{\text{S}}$ 2D systolic arrays to compute Equation 2b. Visually, this module handles data tiles bounded by the dashed lines in Figure 1b. The processing of all the tiles follows the schedule specified by the block matrix multiplication algorithm [20] — i.e., the systolic arrays read tiles of $\widetilde{\boldsymbol{W}}$ along the c_{in} dimension, and then along the c_{out} and n^2 dimensions. The memory layout and the performance analysis are presented later on.

Arithmetic unit: We propose a simple technique to multiply low bit-width complex numbers, using the built-in high bit-width DSPs. Consider the complex multiplication $(p + \text{j}q) \times (m + \text{j}n)$, where p, q are of x-bit and m, n are of y-bit (w.l.o.g., $x \geq y$). As shown in Figure 3, we construct two inputs to the hardware multiplier, one with $(2x + y)$ bits (y bits of zeros between the real part p and the imaginary part q), and the other with $(2y + x)$ bits (x bits of zeros between m and n). The output of the $(2x + y)$-bit multiplier consists of three parts — the first $(x + y)$ bits contain results of $q \cdot n$, the second $(x + y)$ bits contain $p{\cdot}n{+}q{\cdot}m$, and the last $(x + y)$ bits contain $p{\cdot}m$. With an additional $(x + y)$-bit adder to substract $p \cdot m$ and $q \cdot n$, we obtain the real part $(p \cdot m - q \cdot n)$ and the imaginary part $(p \cdot n + q \cdot m)$ of the desired result. In Section III-C, we demonstrate that the above technique enhances the flexibility of our tool in terms of quantization schemes.

Kernel buffer: For large convolutional layers, it may not be feasible to store the entire weight tensor $\widetilde{\boldsymbol{W}}$ on chip. Thus, we perform channel tiling (similar to [3]) to divide the c_{in} and c_{out} dimensions into size-c partitions. The kernel buffer stores a $\widetilde{\boldsymbol{W}}$ tile of shape $c \times c \times n^2$ during inference (Figure 1b). To facilitate data access of the dot product module, we further partition the tile of $\widetilde{\boldsymbol{W}}$ into shape $P_{\text{S}} \times P_{\text{S}} \times N_{\text{S}}$ sub-tiles. Each sub-tile follows the input-channel-major and pixel-major memory layout, so that the BRAMs of the kernel buffer supply $P_{\text{S}} \times N_{\text{S}}$ distinct data per cycle to the systolic arrays.

Input / Output buffer: Consistent with the tiling on $\widetilde{\boldsymbol{W}}$, we divide the c_{in} dimension of $\widetilde{\boldsymbol{X}}$ and c_{out} dimension of $\widetilde{\boldsymbol{Y}}$ into size-c partitions. With batch size b and double buffering, both the input and output buffers are of size $2 \cdot b \cdot c \cdot n^2$ (Figure 1b). Similar to the design of kernel buffer, we further partition the tile of $\widetilde{\boldsymbol{X}}$ and $\widetilde{\boldsymbol{Y}}$ into shape $P_{\text{S}} \times P_{\text{S}} \times N_{\text{S}}$ sub-tiles[2]. Each sub-tile of $\widetilde{\boldsymbol{X}}$ ($\widetilde{\boldsymbol{Y}}$) follows input-channel-major (output-channel-major) and pixel-major memory layout.

Streaming permutation network (SPN): SPNs [19] are included for matrix transformation. SPNs in the 2D FFT module enable streaming processing of the row and column FFTs. The SPNs between 2D FFT and input buffer, and between 2D IFFT and output buffer are essential due to different data layout of these modules. Resource consumption of SPNs is negligible.

C. Design Space Exploration

We use the parameters defined in Section III-A to derive the performance model. Design space exploration is performed to identify mapping parameters[3]: N_{F}, P_{F}, N_{S}, P_{S}, b and c.

Batching constraint: Batch processing is necessary for our dot product based spectral CNN design. Large batch size results in higher BRAM consumption in input and output buffers. Small batch size may cause under-utilization of the systolic arrays. Ideally, batch size should satisfy constraint $\mathbb{C}_0$:

$$\mathbb{C}_0 : \qquad b = P_{\text{S}} \tag{3}$$

DSP constraint: Due to the technique in Figure 3, when the quantization bit width $\widetilde{q}_{\text{act}}$ and $\widetilde{q}_{\text{krn}}$ is low, FPGA can perform more than $\mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}$ number of multiplication per cycle. We define $\widetilde{\mathcal{N}}_{\text{DSP}}^{\text{dot}}$ as the *effective* number of complex multipliers on chip, given the quantization scheme. Further, define $\widetilde{q}' = \max\left\{2\widetilde{q}_{\text{act}} + \widetilde{q}_{\text{krn}}, \widetilde{q}_{\text{act}} + 2\widetilde{q}_{\text{krn}}\right\}$, $\widetilde{q}'' = \max\left\{4\widetilde{q}_{\text{act}} + 5\widetilde{q}_{\text{krn}}, 5\widetilde{q}_{\text{act}} + 4\widetilde{q}_{\text{krn}}\right\}$. So the effective amount of DSP resources is given by[4]: $\widetilde{\mathcal{N}}_{\text{DSP}}^{\text{dot}} = \max_{Q_{\text{P}}}\left\{\left(\frac{1}{3}\mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}\right), \left(\mathbb{1}_{\widetilde{q}' \leq Q_{\text{P}}} \cdot \mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}\right), \left(2 \cdot \mathbb{1}_{\widetilde{q}'' \leq Q_{\text{P}}} \cdot \mathcal{N}_{\text{DSP}}^{Q_{\text{P}}}\right)\right\}$. We thus derive the constraint $\mathbb{C}_1$ due to limited DSPs:

$$\mathbb{C}_1 : \qquad N_{\text{S}} \cdot P_{\text{S}} \cdot P_{\text{S}} \leq \widetilde{\mathcal{N}}_{\text{DSP}}^{\text{dot}} \tag{4}$$

BRAM constraint: Due to data quantization, one row of a BRAM block may store multiple data points of the spectral tensors. Define the effective number of BRAM blocks for the input and output buffers: $\widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{act}} = \frac{1}{2}\left\lfloor\frac{Q_{\text{B}}}{\widetilde{q}_{\text{act}}}\right\rfloor \mathcal{N}_{\text{BRAM}}^{(1)}$; and effective BRAMs for the kernel buffer: $\widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{krn}} = \frac{1}{2}\left\lfloor\frac{Q_{\text{B}}}{\widetilde{q}_{\text{krn}}}\right\rfloor \mathcal{N}_{\text{BRAM}}^{(2)}$. Also $\mathcal{N}_{\text{BRAM}}^{(1)} + \mathcal{N}_{\text{BRAM}}^{(2)} = \mathcal{N}_{\text{BRAM}}^{Q_{\text{B}}}$. Below, $\mathbb{C}_2$, $\mathbb{C}_3$ capture the constraints due to limited BRAMs on-chip. $\mathbb{C}_4$ should be satisfied so that the data parallelism provided by the data buffers are sufficient to keep the systolic arrays busy.

$$\mathbb{C}_2 : \quad 2 \cdot \left(2 \cdot b \cdot c \cdot n^2\right) \leq \mathcal{D}_{\text{BRAM}} \cdot \widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{act}} \tag{5a}$$

$$\mathbb{C}_3 : \quad c \cdot c \cdot n^2 \leq \mathcal{D}_{\text{BRAM}} \cdot \widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{krn}} \tag{5b}$$

$$\mathbb{C}_4 : \quad N_{\text{S}} \cdot P_{\text{S}} \leq \min\left\{\frac{1}{4} \cdot \widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{act}}, \widetilde{\mathcal{N}}_{\text{BRAM}}^{\text{krn}}\right\} \tag{5c}$$

[2]We always choose the batch size $b = P_{\text{S}}$. See Section III-C.

[3]We assume the spectral kernels as the tool's input are quantized based on some given n, so n is not a design parameter. The design space exploration below can be trivially extended to incorporate n as an additional parameter.

[4]*First term*: 3 built-in multipliers perform 1 complex multiplication [21]; *Second term*: 1 built-in multiplier performs 1 complex multiplication (Figure 3); *Third term*: 1 built-in multiplier performs 2 complex multiplications (generalization of Figure 3, applied to extremely low quantization bit width).

Define one *round* as the processing of all data in the input buffer. Total number of clocks (t_{rnd}) to finish one round is:

$$t_{\text{rnd}} = \max\left\{ \frac{2 \cdot b \cdot c \cdot n^2}{\frac{1}{2}\left\lfloor \frac{Q_{\text{D}}}{q_{\text{act}}} \right\rfloor \mathcal{N}_{\text{DRAM}}^{Q_{\text{D}}}},\ \frac{b \cdot c \cdot n^2}{P_{\text{F}} \cdot N_{\text{F}}},\ \frac{(c \cdot c \cdot b) \cdot n^2}{N_{\text{S}} \cdot P_{\text{S}} \cdot P_{\text{S}}} \right\} \tag{6}$$

Total cycles to finish one layer (averaged over batch b) is[5]:

$$t_{\text{lyr}} = \left\lceil \frac{c_{\text{in}}}{c} \right\rceil \cdot \left\lceil \frac{c_{\text{out}}}{c} \right\rceil \cdot \left\lceil \frac{h_{\text{act}}}{n - h_{\text{krn}} + 1} \right\rceil^2 \cdot \frac{t_{\text{rnd}}}{2b} \tag{7}$$

The design space exploration thus solves the below optimization problem, to identify the optimal mapping parameters $(N_{\text{F}}^*, P_{\text{F}}^*, N_{\text{S}}^*, P_{\text{S}}^*, b^*, c^*)$. We constrain all parameters to be power of two. As a reasonable assumption, each of the optimal parameters falls between 2^0 to 2^9, so the total number of design points to be evaluated is less than 10^6. Equation 8 can be solved within one second on a laptop (Intel Core i5).

$$\underset{N_{\text{F}}, P_{\text{F}}, N_{\text{S}}, P_{\text{S}}, b, c}{\text{minimize}} \quad \sum_{\ell=1}^{L} t_{\text{lyr}}^{(\ell)}, \qquad \text{subject to} \quad \mathbb{C}_0, \mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_3, \mathbb{C}_4 \tag{8}$$

Note that the design space exploration algorithm identifies a single design point $(N_{\text{F}}^*, P_{\text{F}}^*, N_{\text{S}}^*, P_{\text{S}}^*, b^*, c^*)$ for all the convolutional layers of the input CNN. Therefore, no runtime reconfiguration is needed during inference.

IV. Experiments

A. *Experimental Setup*

We evaluate our tool[6] on two target FPGAs (Intel Stratix-V GXA7 and Stratix-10 GX2800), using two state-of-the-art CNNs (AlexNet [1] and VGG16 [2]). We implement quantization schemes ranging from 2-bit to 16-bit quantization. The Stratix-10 GX2800 FPGA is suitable for data center workloads. It has 229 Mb BRAM, 5760 DSPs and 3,732,480 ALMs. Each DSP supports either one 27-bit or two 18-bit fixed point multiplications. Stratix-V GXA7 has 50 Mb BRAM, 256 DSPs and 234,720 ALMs. We use Quartus Prime Pro 18.1 for synthesis. The results are post place-and-route results.

Our metric is inference throughput, measured by images-per-second. We adopt a CPU-FPGA co-processing model. The spectral convolution layers are executed on FPGA, and all other layers (e.g., ReLU, pooling, fully-connected) are executed by CPU. Also, for both AlexNet and VGG16, since the first convolutional layer has only three input channels (R.G.B. color channels), the channel tiling mechanism may not achieve high efficiency on FPGAs. Thus, we execute the first convolutional layer of the two CNNs on CPU. Under such a schedule, for AlexNet and VGG16, CPU is responsible for less than 15% and 2% of the total computation respectively. While our tool supports spectral CNNs with any FFT size n, in all the experiments below, we keep n to be the optimal value (i.e., 16) from the design space exploration in [3].

[5]The factor 2 in denominator is due to the technique in [3] to feed both real and imaginary channels at FFT input with spatial activations.

[6]Open sourced at: https://github.com/ZimpleX/FPGA-Spec-CNN-FPL19

B. *Flexibility w.r.t. Quantization Scheme*

Since the DSPs of the Stratix family only supports 18-bit or 27-bit fixed-point multiplication, the arithmetic unit design proposed in Section III-B can efficiently improve throughput when the CNN is quantized to low bit-width data. Figure 4 shows the number of complex multiplications supported by DSPs of Stratix-10 GX2800 under various bit widths[7]. Reduced quantization bit width leads to increased number of multiplications, and thus higher throughput on a given device.

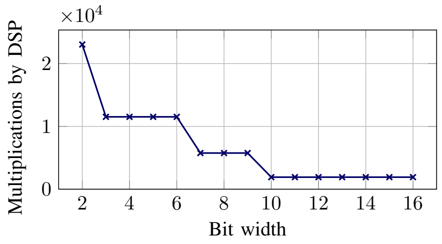


Fig. 4: Number of complex multiplications *vs.* bit width

C. *Flexibility w.r.t. CNN Models*

CNN parameters h_{act}, h_{krn}, c_{in} and c_{out} vary significantly for convolutional layers within and across CNN models. Accelerators generated by our tool can sustain high throughput under most of the model parameter settings. Specifically, the variation in the activation size h_{act} is handled by the Overlap-Add technique in software [3], [4]. The variation in the spatial kernel size h_{krn} is inherently resolved by the spectral convolution algorithm (since the spectral kernel size is n, regardless of h_{krn}). The variation in the number of channels is handled by the tiling technique described in Section III-B.

Figure 5 shows the performance breakdown for each convolutional layer. The measured throughput of each layer is normalized by the theoretical peak throughput (the ideal throughput when all DSPs on the chip are 100% utilized). The generated accelerators achieve close to peak throughput for most convolutional layers. Throughput of layer 2 (AlexNet) and layers 2,3 (VGG16) are relatively low on Stratix-10. The accelerators implemented on FPGAs with massive DSP and BRAM resources often require larger data tiles, which may not be fully filled by initial layers with small number of channels.

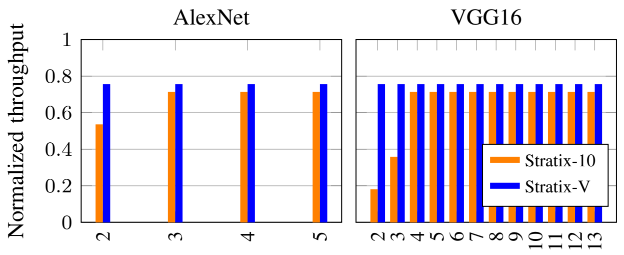


Fig. 5: Performance breakdown by convolutional layers

[7]For presentation conciseness, we assume $q_{\text{act}} = \widetilde{q}_{\text{act}} = \widetilde{q}_{\text{krn}}$.

TABLE I: Comparison with state-of-the-art AlexNet and VGG16 implementations

	AlexNet					VGG16				
	[3]	[9]	[9]	Proposed	Proposed	[3]	[9]	[9]	Proposed	Proposed
FPGA	Stratix-10 GX2800	UltraScale KU115	UltraScale KU115	Stratix-10 GX2800	Stratix-10 GX2800	Stratix-10 GX2800	UltraScale KU115	UltraScale KU115	Stratix-10 GX2800	Stratix-10 GX2800
Frequency (MHz)	120	220	220	200	200	120	235	235	200	200
Quantization	16-bit	16-bit	8-bit	16-bit	8-bit	16-bit	16-bit	8-bit	16-bit	8-bit
DSP Usage	3264 (56%)	4854 (88%)	4854 (88%)	3264 (56%)	4480 (78%)	3264 (56%)	4318 (78%)	4318 (78%)	3264 (56%)	4480 (78%)
Logic Usage	413K (45%)	262K (40%)	262K (40%)	140K (15%)	150K (16%)	419K (47%)	258K (39%)	258K (39%)	140K (15%)	150K (16%)
BRAM blocks	6129 (52%)	986 (46%)	986 (46%)	2616 (22%)	5232 (45%)	6133 (52%)	1578 (81%)	1578 (81%)	2616 (22%)	5232 (45%)
Throughput (img/sec)	1704	1126	2252	2841	9114	77	65	130	129	308

D. Flexibility w.r.t. FPGA Architecture

To show flexibility of the tool we did design space exploration by placing constraints on the amount of DSPs and BRAMs that can be used by the accelerator. In Figure 6,

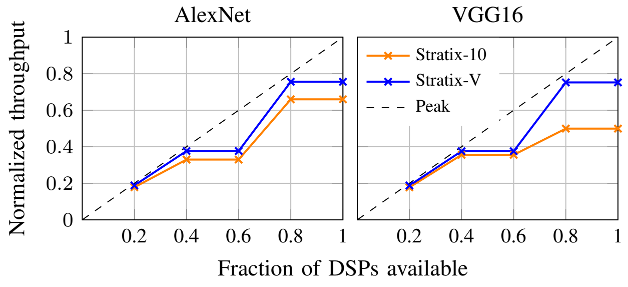


Fig. 6: Throughput *vs.* limited DSP resources

we allow the tool to use all BRAM resources and only restrict the amount of DSPs available. Again, we normalize the throughput by the theoretical peak throughput when 100% of DSPs are available. Ideally, the peak throughput should be proportional to the amount of DSPs available (dashed line). On Stratix-V, the measured throughput matches well with the peak throughput. On Stratix-10 with VGG16, measured throughput is over 50% of the ideal throughput when 80% to 100% of the total DSPs are available. In such cases, the external DRAM does not have high enough bandwidth to match the computation speed of DSPs. In Figure 7, we allow

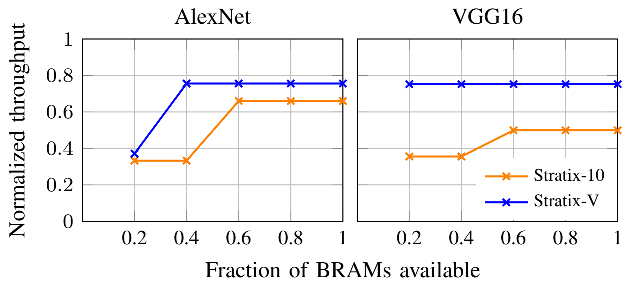


Fig. 7: Throughput *vs.* limited BRAM resources

the tool to use all DSPs, but limit the number of M20K BRAM blocks. Due to data tiling and our BRAM efficient architecture based on streaming systolic array, the overall throughput is not sensitive to the amount of available BRAMs. Throughput degradation is only observed when the amount of allowed BRAMs is restricted to less than 50% of the required BRAMs.

E. Comparison with State-of-the-Art

Table I summarizes the comparison with other design automation tools. [3] is the state-of-the-art tool for spectral CNNs, and [9] is the state-of-the-art tool for spatial CNNs. Both UltraScale KU115 and Stratix-10 GX2800 are high-end devices with about the same amount of resources. Comparing with [9], we achieve 2.5× (16-bit) and 4.0× (8-bit) higher throughput on AlexNet, and 2.0× (16-bit) and 2.4× (8-bit) higher throughput on VGG16. The throughput improvement is due to the low complexity of spectral convolution algorithm, as well as the highly optimized systolic array-based design. Note that the designs generated by our tool consume 1) small amount of logic resources due to the resource efficient systolic array pipeline, and 2) not much BRAM resources, which is consistent with the evaluation in Section IV-D. Also note that [9] can generate designs with low latency. However, latency optimization is not the focus of this paper. As for [3], since only results on Stratix-V devices are available in the original paper, we re-implement its design on Stratix-10 for a fair comparison with our design. Clearly, [3] requires much more routing resources for BRAM connection to support the DSP computation. Thus, its parallelization strategy results in severe clock rate degradation (120 MHz) on Stratix-10. On the other hand, our designs can maintain high clock rate (200 MHz), because the streaming nature of systolic arrays significantly reduces the required number of BRAM reads/writes per cycle.

Note that the DSP consumption in our design is dominated by the dot product module and not by the FFT / IFFT modules. For the 16-bit design, 3072 (94%) DSPs are used to perform dot product, and 192 (6%) are used to perform FFT / IFFT. For the 8-bit design, 4096 (91%) DSPs are used to perform dot product, and 384 (9%) are used to perform FFT / IFFT.

V. Conclusion

We presented a flexible design automation tool to generate high throughput inference accelerators for quantized, spectral CNNs. We demonstrated the flexibility of the tool in terms of CNN models, FPGA architectures and quantization schemes.

In the future we plan to extend the tool to support hybrid processing of spatial and spectral convolutional layers, under a *unified* systolic array-based architecture. As spectral convolution may not be beneficial on 1×1 kernels in CNNs such as [22]–[25], it is justifiable to compute the layers requiring 1×1 kernels by the native spatial convolution. We will also develop spectral quantization algorithms to incorporate accuracy evaluation into the tool.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS'12*, 2012.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[3] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput cnn implementations on fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018.

[4] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021727

[5] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with fft on embedded hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737–1749, Sep. 2018.

[6] W. Sun, H. Zeng, Y. E. Yang, and V. Prasanna, "Throughput-optimized frequency domain cnn with fixed-point quantization on fpga," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2018, pp. 1–8.

[7] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 29:1–29:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062207

[8] H. Zeng, C. Zhang, and V. Prasanna, "Fast generation of high throughput customized deep learning accelerators on fpgas," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2017, pp. 1–8.

[9] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 56:1–56:8. [Online]. Available: http://doi.acm.org/10.1145/3240765.3240801

[10] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–8.

[11] "Intel stratix 10," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf, accessed: 2019-03-12.

[12] C. Zhuge, X. Liu, X. Zhang, S. Gummadi, J. Xiong, and D. Chen, "Face recognition with hybrid efficient convolution algorithms on fpgas," *CoRR*, vol. abs/1803.09004, 2018. [Online]. Available: http://arxiv.org/abs/1803.09004

[13] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[14] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 163–1636.

[15] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[16] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 15–24.

[17] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 152–159.

[18] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[19] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 240–249. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689068

[20] G. Nimako, E. J. Otoo, and D. Ohene-Kwofie, "Fast parallel algorithms for blocked dense matrix multiplication on shared memory architectures," in *Algorithms and Architectures for Parallel Processing*, Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Zomaya, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 443–457.

[21] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[23] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[24] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[25] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.